三、多路复用IO

2024年6月29日 20:42

多进程服务器

每当有一个新的客户端建立连接，就会创建一个新的进程为这个客户端服务
当某一个客户端断开连接时，子进程终止

问题：

- 频繁的创建进程和销毁进程，系统开销较大
- 能够承载的上限低
- 能否实现一个进程就可以和多个客户端进行通信？（多路IO）

三、多路复用IO

3.1 多路复用IO的概念和作用

多路复用 I/O(Multiplexing I/O)是一种 I/O 模型，用于处理多个 I/O 操作，同时允许程序等待多个输入或输出事件而不会阻塞。它在网络编程中扮演着重要角色，可以提高程序的并发性能和效率

多路复用 I/O 的主要目的是使程序能够同时监听多个文件描述符(通常是套接字)，在有事件发生时立即做出响应，而不需要在等待一个套接字上的 I/O 完成时阻塞整个程序。这种模型有助于避免创建大量线程或进程来处理并发连接，从而节省系统资源并提高程序的性能。

多路复用 1/0 的作用包括:

- 提高并发性能：多路复用 1/0 允许一个线程或进程同时监听多个套接字上的1/0 事件，从而使程序能够同时处理多个连接。
- 减少资源消耗：相比创建大量线程或进程来处理并发连接，多路复用 I/0可以节省系统资源，减少上下文切换的开销。
- 避免阻塞：多路复用 I/0 允许程序在等待 I/0 事件的同时继续执行其他任务，避免了阻塞整个程序。
- 简化编程模型：多路复用 I/0 可以将不同套接字的 I/0 事件汇总到一个地方，简化了编程模型，使代码更加清晰易懂。

常见的多路复用 I/0 模型包括 select、poll、epoll(在 Linux 中)，它们在不同操作系统和环境中具有类似的功能，但可能有不同的性能和用法。多路复用 IO 在服务器编程中经常用于监听多个客户端连接，实现高并发的网络服务。

多路IO复用：几个特殊的函数（select、poll、epoll）

多路IO复用解决了什么问题？

lfd、cfd1、cfd2、cfd3（一个服务器、三个客户端）
他们的读事件什么时候发生，无法确定，因此最开始我们没做任何处理，导致多个客户端连接服务器之后，服务器是读不到消息的。无法实现相关功能。
之后，我们利用多进程（多线程）处理不同的客户端申请连接，但是开销太大。

多路IO：帮助我们在一个进程（线程）下， 一起监听很多个fd的事件（有客户端申请连接或是客户端发送消息），当有事件触发的时候，可以及时的通知用户处理。（多路IO会选择合适的时机去调用accept或者是read，保证不会阻塞）

3.2 select()系统调用

函数描述
对于lfd、cfd1、cfd2
select可以帮助我们监听lfd、cfd1、cfd2的事件，当其中一个或多个事件发生时，select可以立刻通知用户，并告知是那个文件描述符的那个事件发生了。

lfd-->事件触发--->accept()

cfd1-->读事件触发--->read(cfd1)
cfd2-->读事件触发--->read(cfd2)

头文件：

```
#include <sys/select.h>
```

函数原型：

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

辅助宏函数

```
void FD_CLR(int fd, fd_set *set);   // 将fd从set集合中清除（把某一位置0）
int  FD_ISSET(int fd, fd_set *set); // 测试fd是否在set集合中（判断某一位是0还是1）
void FD_SET(int fd, fd_set *set);   // 将fd加入set集合（把某一位置1）
void FD_ZERO(fd_set *set);          // 将set清零使集合中不含任何fd（全部置零）
```

函数参数：

- int nfds // 说明fd_set 用到了的第几位（监听的最大的文件描述符+1）
- fd_set *readfds // 文件描述符的集合，监听读事件的文件描述符集合(传入传出参数)
- fd_set *writefds // 文件描述符的集合，监听写事件的文件描述符集合
- fd_set *exceptfds // 文件描述符的集合，监听异常事件的文件描述符集合
- struct timeval *timeout // 超时事件，select只等待这个时间，没有事件发生就会返回

位图 fd_set 的本质就是一个长整型（long）的数组。大小为128字节（一个long类型8个字节，因此是16个），共1024位，每一位都能表示一个文件描述符，bits[1024],因此一个fd_set可以存0-1023的文件描述符,

readfds、writefds、exceptfds 传入传出参数
以readfds为例，如果监听到fd为2，3的文件描述符触发读事件，select就会将readfds的集合设置为[0,0,1,1,0...0]

struct timeval *timeout Linux时间的一个结构体，结构体中都是长整型，精确到了秒和微秒

```
struct timeval {
    time_t      tv_sec;         /* seconds */ 秒
    suseconds_t tv_usec;        /* microseconds */ 微秒
};
        // select只等待 5.01秒。没有事件发生就返回
        // timeout是一个传入传出的参数，传出的是剩余的时间，
        // 没有事件发生，剩余0了，就查看一下有没有发生，就一点也不等了，
        // 因此要在循环内设置时间
```

select()的返回值

作为函数的返回值，select()会返回如下几种情况中的一种。

- 返回 -1 表示有错误发生。可能的错误码包括 EBADF 和 EINTR。EBADF 表示 readfdswritefds 或者 exceptfds 中有一个文件描述符是非法的(例如当前并没有打开)。EINTR表示该调用被信号处理例程中断了。(如果被信号处理例程中断，select()是不会自动恢复的。
- 返回 0表示在任何文件描述符成为就绪态之前 select()调用已经超时。在这种情况下，每个返回的文件描述符集合将被清空
- 返回一个正整数表示有1个或多个文件描述符已达到就绪态。返回值表示处于就绪态的文件描述符个数。在这种情况下，每个返回的文件描述符集合都需要检查(通过FD_ISSET())，以此找出发生的 I/O 事件是什么。

如果同一个文件描述符在 readfds、writefds 和 exceptfds 中同时被指定，且它对于多个 I/O 事件都处于就绪态的话，那么就会被统计多次。换句话说，select()返回所有在3个集合中被标记为就绪态的文件描述符总数。

```
// 端口号
#define SOCKPORT 8001
// select 简单实现
```

```c
int main(int argc, char* argv[])
{
    int lfd = socket(AF_INET,SOCK_STREAM,0);
    if(lfd < 0)
    {
        perror("socket error");
        exit(1);
    }
    // 绑定
    struct sockaddr_in serArr;

    serArr.sin_port = htons(SOCKPORT);
    serArr.sin_family = AF_INET;
    serArr.sin_addr.s_addr = INADDR_ANY;
    int bret =  bind(lfd,(struct sockaddr*)&serArr,sizeof(serArr));
    if(bret < 0)
    {
        perror("bind error");
        exit(1);
    }
    // 监听
    listen(lfd,64);
    printf("listening...\n");

    struct sockaddr_in cliArr;
    int len = sizeof(cliArr);
    // 设置读集合
    fd_set read_set,all_set;
    FD_ZERO(&all_set);
    FD_SET(lfd,&all_set);
    int maxfd = lfd; //当前最大的文件描述符
    char buf[1024];

    struct timeval time;

    while(1)
    {
        // 第一次只监听lfd，如果没有触发事件，select会一直阻塞
        read_set = all_set; // all_set是所有需要监听的文件描述符，read_set是触发事件的文件描述符
        int sret = select(maxfd+1,&read_set,NULL,NULL,&time);
        if(sret ==-1)
        {
            perror("select error");
            exit(1);
        }
        // lfd的读事件是否发生
        if(FD_ISSET(lfd,&read_set) == 1)
        {
            int cfd = accept(lfd,(struct sockaddr*)&cliArr,&len);
            if(cfd<0)
            {
                perror("accept error");
                exit(1);
            }
            FD_SET(cfd,&all_set);
            if(maxfd<cfd)
            {
                maxfd = cfd;
            }

            int port = ntohs(cliArr.sin_port);
            char bst[64];
            inet_ntop(AF_INET,&cliArr.sin_addr.s_addr,bst,sizeof(bst));
            printf("accept successful!\n");
            printf("client IP: %s\n",bst);
            printf("client Port: %d\n",port);
            if(--sret == 0)
            {
                continue;
            }
        }
        for(int i = lfd+1;i<maxfd+1;i++)
        {
            if(FD_ISSET(i,&read_set) == 1)
            {
                int read_count = read(i,buf,sizeof(buf));
                write(i,buf,read_count);
                write(STDOUT_FILENO,buf,read_count);
                if(--sret == 0)
```

```
                {
                    break;
                }
            }
        }
    }
    return 0;
}
```

select解决了什么问题？
select、poll、epoll的对比？

select的优缺点

优点：

- 比较古老、跨平台

缺点：（面试）

- 文件描述符的监听上限（1024个）；
- 频繁调用系统调用（如果有1000个客户端，每个客户端发10句话，就会进行1w次系统调用），就会产生用户态空间和内核态空间的大量数据拷贝；
- 轮询监听（select的监听是轮询的，如果在某一时刻，某个文件描述符触发事件，是无法立即处理的，需要轮询到才能处理，导致这个客户端需要花费时间等待。如果轮询了一遍，只有几个事件触发，这一遍轮询就浪费了）

  当监听数量多时，效率较低，延时较大

  当活跃用户少时，效率较低，浪费资源

系统调用：**预读入缓输出**

通过系统调用，每次写一个字符利用系统调用写1000次 和 调用一次系统调用写1000个字符

肯定是第二个好，而我们在通过系统调用写数据时，也是这样的，系统调用不会立即写入，而是找一个合适的时间一次性写入。

称为预读入缓输出

因此在频繁写入的时候，尽量减少系统调用，使用库函数（一般都有缓冲区，满了才调用系统调用写入）

使用 strace 查看系统调用次数

```
// 预读入缓输出验证
// 使用库函数
int main()
{
    FILE* stream = fopen("./m.txt","w+");
    for(int i = 0; i <10000; i++)
    {
        fputc(1,stream);
    }
    return 0;
}
```

```
newfstatat(3, "", {st_mode=S_IFREG|0664, st_size=0, ...}, AT_EMPTY_PATH) = 0
write(3, "\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1"..., 4096) = 4096
write(3, "\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1"..., 4096) = 4096
write(3, "\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1\1"..., 1808) = 1808
exit_group(0)                           = ?
+++ exited with 0 +++
weihong@weihong:~/网络编程/day_630/select$
```

只调用了 3 次系统调用

```
// 使用系统调用
int main()
{
    int fd = open("./m.txt",O_RDWR);
    for(int i = 0; i <10000; i++)
    {
        write(fd,"1",1);
```

```
    }
    return 0;
}
```

```
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
write(3, "1", 1)                     = 1
exit_group(0)                        = ?
+++ exited with 0 +++
```

调用了1w次系统调用

3.3 poll()系统调用

系统调用 poll()执行的任务同 select()很相似。两者间主要的区别在于我们要如何指定待检查的文件描述符。在 select()中，我们提供三个集合，在每个集合中标明我们感兴趣的文件描述符。而在 poll()中我们提供一列文件描述符，并在每个文件描述符上标明我们感兴趣的事件

函数原型

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

函数参数

参数 fds 列出了我们需要 poll()来检查的文件描述符。该参数为 polfd 结构体数组，其定义如

```
struct pollfd {
        int   fd;          /* file descriptor */
        short events;      /* requested events */
        short revents;     /* returned events */
    };
```

参数 nfds 指定了数组 fds 中元素的个数。数据类型 nfdst实际为无符号整形。

pollfd 结构体中的 events 和 revents 字段都是位掩码。

调用者初始化 events 来指定需要为描述符 fd 做检查的事件。

当 pol()返回时，revents 被设定以此来表示该文件描述符上实际发生的事件。

下表列出了可能会出现在 events 和 revents 字段中的位掩码。该表中第一组位掩码(POLLIN、POLLRDNORM、POLLRDBAND、POLLPRI以及 POLLRDHUP)同输入事件相关。下一组位掩码(POLLOUT、POLLWRNORM 以及 POLLWRBAND)同

输出事件相关。第三组位掩码(POLLERR、POLLHUP 以及 POLLNVAL)是设定在revents 字段中用来返回有关文件描述符的附加信息。如果在 events 字段中指定了这些位掩码，则这三位将被忽略。在 Linux 系统中，poll()不会用到最后一个位掩码 POLLMSG。

位 掩 码	events 中的输入	返回到 revents	描 述
POLLIN	●	●	可读取非高优先级的数据
POLLRDNORM	●	●	等同于 POLLIN
POLLRDBAND	●	●	可读取优先级数据（Linux 中不使用）
POLLPRI	●	●	可读取高优先级数据
POLLRDHUP	●	●	对端套接字关闭
POLLOUT	●	●	普通数据可写
POLLWRNORM	●	●	等同于 POLLOUT
POLLWRBAND	●	●	优先级数据可写入
POLLERR		●	有错误发生
POLLHUP		●	出现挂断
POLLNVAL		●	文件描述符未打开
POLLMSG			Linux 中不使用（SUSv3 中未指定）

如果我们对某个特定的文件描述符上的事件不感兴趣，可以将 events 设为 0。另外，给 fd 字段指定一个负值(例如，如果值为非零，取它的相反数)将导致对应的 events 字段被忽略目 revents 字段将总是返回 0。这两种方法都可以用来(也许只是暂时的)关闭对单个文件描述符的检查，而不需要重新建立整个 fds 列表。

注意，下面进一步列出的要点主要是关于 poll()的 Linux 实现。

1.尽管被定义为不同的位掩码，POLLIN 和 POLLRDNORM 是同义词。

2.尽管被定义为不同的位掩码，POLLOUT和 POLLWRNORM 是同义词，

3.一般来说 POLLRDBAND 是不被使用的，也就是说它在 events 字段中被忽略，也不会设定到revents 中去

poll()真正关心的标志位就是 POLLIN、POLLOUT、POLLPRI、POLLRDHUP、POLLHUP 以及 POLLERR。

timeout 参数

参数 timeout 决定了 poll()的阻塞行为，单位是毫秒,具体如下。

1.如果 timeout 等于 -1，pol(l)会一直阻塞直到 fds 数组中列出的文件描述符有一个达到就绪态(定义在对应的 events 字段中)或者捕获到一个信号。

2.如果 timeout 等于 0，poll()不会阻塞一只是执行一次检查看看哪个文件描述符处于就绪态。

3.如果 timeout 大于0，poll()至多阻塞 timeout 毫秒，直到 fds 列出的文件描述符中有一个达到就绪态，或者直到捕获到一个信号为止。

poll()的返回值

作为函数的返回值，poll()会返回如下几种情况中的一种。

1、返回 -1 表示有错误发生。一种可能的错误是 EINTR，表示该调用被一个信号处理例程中断。(如果被信号处理例程中断，poll()绝不会自动恢复。)

2、返回 0表示该调用在任意一个文件描述符成为就绪态之前就超时了

3、返回正整数表示有1个或多个文件描述符处于就绪态了。返回值表示数组 fds 中拥有非零revents 字段的 pollfd 结构体数量。

注意 select()同 pol()返回正整数值时的细小差别。如果一个文件描述符在返回的描述符集合中出现了不止一次，系统调用 select()会将同一个文件描述符计数多次。而系统调用poll()返 回的是就绪态的文件描述符个数，且一个文件描述符只会统计一次，就算在相应的revents 字段中设定了多个位掩码也是如此。

poll的优缺点

优点：

文件描述符无上限

缺点：

结构体大小为8个字节，每一个文件描述符都需要8个字节，1024个文件描述符就需要 8192个字节，频繁调用的话，在内核中就会进行大量的数据拷贝，浪费时间，影响用户体验

轮询监听，

3.4 epoll()系统调用

epoll(),linux独有的，监听百万级的数据量。

前面说了select和poll的缺点，
- 文件描述符的监听上限（1024个，poll无上限，但所需空间更大）；
- 用户空间和内核空间的大量数据拷贝；
- 轮询监听

那么epoll是如何解决这些问题的呢？

epoll将他的行为分成了三个函数：epoll_create、epoll_ctl、epoll_wait

epoll_create：在内核空间创建一个epoll对象

这个对象包含一个存储所有待监听的文件描述符的结构体--使用了红黑树（减少增删改查的时间复杂度）
一个存储所有触发了事件的文件描述符--使用的是链表

利用了终端原理（callback机制），黑红树中，一旦有文件描述符触发事件，就会利用回调机制，添加到链表中

这两个数据结构是一直存储在内核中的，触发一个，回调一个，触发一个，回调一个，而不是像（select、和poll）触发一个，将所有的文件描述符都返回，在调用epoll的时候也一样，调用一个，就向黑红树中插入一个，而不是像（select、poll）一样，每次调用都会将所有的文件描述符传入。
这样就减少了大量的用户空间到内核空间的拷贝。

虽然epoll可以监听的数据量很大，但他的内核开销是很大的，如果客户端的访问请求很小的话，select也是很好的选择

对于epoll来讲，虽然它可以监听百万级别的数据量，但他针对的是大多数用户是不活跃的状态，如果百万级的用户同时触发事件（高并发），他也不好处理（和轮询也就没啥区别了），但事实上，在日常生活中，一个服务器的活跃人数是不会很多的，不活跃人数很多。

同 1/0 select和 pol 一样，Linux的 epoll(event poll)API也可以检查多个文件描述符上的I/0 就绪状态。epol API的主要优点如下。

1.当检查大量的文件描述符时，epol 的性能延展性比 select()和 pol()高很多
2.poll API既支持水平触发也支持边缘触发。与之相反，select()和 pol()只支持水平触发，而信号驱动 I/0 只支持边缘触发

epoll API是 Linux 系统专有的，在 2.6 版中新增。

epoll API的核心数据结构称作 epoll 实例，它和一个打开的文件描述符相关联。这个文件描述符不是用来做 I/0 操作的，相反，它是内核数据结构的句柄，这些内核数据结构实现了两个目的。

1.记录了在进程中声明过的感兴趣的文件描述符列表 -interest list(兴趣列表)。
2.维护了处于 I/0 就绪态的文件描述符列表 -ready list(就绪列表)。

ready list 中的成员是 interest list 的子集。

对于由 epol 检查的每一个文件描述符，我们可以指定一个位掩码来表示我们感兴趣的事件。这些位掩码同 pol()所使用的位掩码有着紧密的关联。

epoll API由以下3个系统调用组成

1、系统调用 epoll create()创建一个 epol 实例，返回代表该实例的文件描述符。

2、系统调用 epoll_ctl()操作同 epoll 实例相关联的兴趣列表。通过 epoll_ctl()，我们可以增加新的描述符到列表中，将已有的文件描述符从该列表中移除，以及修改代表文件描述符上事件类型的位掩码。

3、系统调用 epoll wait()返回与 epol 实例相关联的就绪列表中的成员，

3.4.1 epoll_create 创建epoll

```
#include <sys/epoll.h>
```

函数原型：

```
int epoll_create(int size);
```

函数参数：

2.6之后size就没什么作用了，之前是需要内核提前申请多大的内存空间，现在内核会自己维护，不够了他自己会申请

返回值：

一个文件描述符，能够让我们在内核中找到这个epoll对象

3.4.2 epoll_ctl 控制epoll对象，修改epoll的兴趣列表

函数原型：

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

函数参数

- **int epfd**：你要控制的那个epoll对象，也就是create返回的文件描述符

  参数 fd 指明了要修改兴趣列表中的哪个文件描述符的设定。该参数可以是代表管道、FIFO、套接字、POSIX 消息队列、终端、设备，甚至是另一个 epoll 实例的文件描述符(例如，我们可以为受检查的描述符建立起一种层次关系)。但是，这里fd 不能作为普通文件或目录的文件描述符(会出现 EPERM 错误)。

- **int op**：用来操作红黑树的参数（增加节点、删除节点，修改节点监听的事件）

  参数 op 用来指定需要执行的操作，它可以是如下几种值

  EPOLL_CTL_ADD

  将描述符 fd 添加到 epoll 实例 epfd 中的兴趣列表中去。对于 fd 上我们感兴趣的事件，都指定在 ev 所指向的结构体中，下面会详细介绍。如果我们试图向兴趣列表中添加一个已存在的文件描述符，epoll_ctl()将出现 EEXIST 错误

  EPOLL_CTL_MOD

  修改描述符 fd 上设定的事件，需要用到由 ev 所指向的结构体中的信息。如果我们试图修改不在兴趣列表中的文件描述符，epoll_ctl()将出现 ENOENT 错误。

  EPOLL_CTL_DEL

  将文件描述符 fd 从 epfd 的兴趣列表中移除。该操作忽略参数 ev。如果我们试图移除一个不在epfd 的兴趣列表中的文件描述符，epoll_ctl()将出现 ENOENT 错误。关闭一个文件描述符会自动将其从所有的 epol 实例的兴趣列表中移除。

- **int fd**：你要监听的那个文件描述符

- **struct epoll_event *event**

```
struct epoll_event {
              uint32_t     events;      /* Epoll events */
              epoll_data_t data;        /* User data variable */
          };
```

1、结构体 epoll _event 中的 events 字段是一个位掩码，它指定了我们为待检查的描述符 fd 上所感兴趣的事

件集合。我们将在下一节中说明该字段可使用的掩码值。

2、data 字段是一个联合体，当描述符 fd 稍后成为就绪态时，联合体的成员可用来指定传回给调用进程的信息。

```
typedef union epoll_data {
    void        *ptr; // 不用
    int          fd;  // 传epoll_ctl第三个参数的fd
    uint32_t     u32; // 不用
    uint64_t     u64; // 不用
} epoll_data_t;
```

3.4.3 epoll_wait

函数描述

系统调用 epoll_wait()返回 epoll 实例中处于就绪态的文件描述符信息。单个 epoll_wait()调用能返回多个就绪态文件描述符的信息。

如果**events链表**为空，而timeout设置了阻塞，wait就会一直等待（或等待阻塞时间）

函数原型

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

函数参数

- **int epfd：**要关注的是哪个epoll对象，即epoll_create的返回值

- **struct epoll_event *events：**一个传出参数，需要自己定义，

  参数 events 所指向的结构体数组中返回的是有关就绪态文件描述符的信息。(结构体epoll_event已经在上一节中描述。)数组 events 的空间由调用者负责申请，所包含的元素个数在参数 maxevents 中指定:

  在数组 events 中，每个元素返回的都是单个就绪态文件描述符的信息。events 字段返回了在该描述符上已经发生的事件掩码。Data 字段返回的是我们在描述符上使用cpollctl()注册感兴趣的事件时在 ev.data 中所指定的值。注意，data 字段是唯一可获知同这个事件相关的文件描述符号的途径。因此，当我们调用 epollctl()将文件描述符添加到兴趣列表中时，应该要么将ev.data.fd 设为文件描述符号，要么将 ev.data.ptr 设为指向包含文件描述符号的结构体。

- **int maxevents：**自己定义的events有多大，就传多大，为了防止上面的数组越界。
- **int timeout：** 用来确定 epoll_wait()的阻塞行为，有如下几种。

  如果 timeout 等于 -1，调用将一直阻塞，直到兴趣列表中的文件描述符上有事件产生，或者直到捕获到一个信号为止。

  如果 timeout 等于0，执行一次非阻塞式的检查，看兴趣列表中的文件描述符上产生了哪个事件。

  如果 timeout 大于 0，调用将阻塞至多 timeout 毫秒，直到文件描述符上有事件发生，或者直到捕获到一个信号为止。

函数返回值

调用成功后，epoll_wait()返回数组 evlist 中的元素个数。
如果在 timeout 超时间隔内没有任何文件描述符处于就绪态的话，返回 0。
出错时返回 -1，并在 errno 中设定错误码以表示错误原因。

在多线程程序中，可以在一个线程中使用 epol_ctl()将文件描述符添加到另一个线程中由 epoll_wait()所监视的 epoll 实例的兴趣列表中去。这些对兴趣列表的修改将立刻得到处理，而epoll_wait()调用将返回有关新添加的文件描述符的就绪信息。

如果自己定义的**struct epoll_event *events** 空间较小，触发的事件较多，存不下所有触发了事件的文件描述符，剩下的就会等到下一趟进行返回。

epoll实现

```
// 端口号
#define SOCKPORT 8001
// epoll 简单实现
int main(int argc, char* argv[])
{
    int lfd = socket(AF_INET,SOCK_STREAM,0);
    if(lfd < 0)
    {
        perror("socket error");
        exit(1);
    }
    // 绑定
    struct sockaddr_in serArr;

    serArr.sin_port = htons(SOCKPORT);
    serArr.sin_family = AF_INET;
    serArr.sin_addr.s_addr = INADDR_ANY;
    int bret =  bind(lfd,(struct sockaddr*)&serArr,sizeof(serArr));
    if(bret < 0)
    {
        perror("bind error");
        exit(1);
    }
    // 监听
    listen(lfd,64);
    printf("listening...\n");

    struct sockaddr_in cliArr;
    int len = sizeof(cliArr);
    // 用来读取的缓冲区
    char buf[1024];
    // ****** epoll ******

    // 先创建一个epoll
    int epfd = epoll_create(64);
    if(epfd == -1)
    {
        perror("epoll_create error");
    }
    // 修改epoll -- 添加lfd
    // int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
    struct epoll_event event; // 创建一个结构体，用来传入ctl
    event.events = EPOLLIN; // 表示监听读事件
    event.data.fd = lfd;    // 传入文件描述符，方便后续判断
    epoll_ctl(epfd,EPOLL_CTL_ADD,lfd,&event);
    // 创建传入 epoll_wait的结构体数组
    struct epoll_event events[1025];
    while(1)
    {
        // 创建事件等待函数，timeout设置为-1，就绪链表中没有事件，一直阻塞
        // timeout 设置为 500，阻塞500毫秒后返回
        int wret = epoll_wait(epfd,events,1025,-1); // 返回所有就绪态的事件个数
        if(wret ==-1)
        {   // epoll_wait返回-1，出错提示
            perror("epoll_wait error");
            exit(1);
        }
        else if(wret == 0)
        {   // epoll_wait返回0，代表阻塞500毫秒期间，没有事件触发
            printf("epoll_wait ret = %d\n",wret);
            continue;
        }
        else
        {
            // 处理wret大于0的情况，即有事件触发
            // wret记录的就是触发事件的个数，因此我们直接遍历wret即可，不用遍历1025个
            for(int i = 0; i < wret;i++)
```

```
            {
                if(events[i].data.fd == lfd) // 如果是lfd触发的事件，需要进行accept连接
                {
                    if(events[i].events & EPOLLIN) // 判断是否为lfd的读事件发生，这里使用按位与，而不是==
                    {
                        int cfd = accept(lfd,(struct sockaddr*)&cliArr,&len); // 创建新的文件描述符
                        if(cfd < 0)
                        {
                            perror("accept error");
                            exit(1);
                        }
                        // 打印建立连接的客户端ip地址和端口号
                        int port = ntohs(cliArr.sin_port);
                        char bst[64];
                        inet_ntop(AF_INET,&cliArr.sin_addr.s_addr,bst,sizeof(bst));
                        printf("accept successful!\n");
                        printf("client IP: %s\n",bst);
                        printf("client Port: %d\n",port);
                        // 将新连接的客户端添加到epoll对象中。
                        event.data.fd = cfd;
                        event.events = EPOLLIN;
                        epoll_ctl(epfd,EPOLL_CTL_ADD,cfd,&event);
                    }
                }
                else // cfd，客户端触发事件
                {
                    if(events[i].events & EPOLLIN) // 判断是否为客户端的读事件触发
                    {
                        int read_count = read(events[i].data.fd,buf,sizeof(buf));
                        if(read_count < 0)
                        {   // 读取失败
                            perror("read error");
                            exit(1);
                        }
                        else if(read_count == 0)
                        {
                            // 表示客户端断开连接了
                            printf("客户端已断开连接\n");
                            // 将该文件描述符对应的epoll对象删除
                            epoll_ctl(epfd,EPOLL_CTL_DEL,events[i].data.fd,NULL); // 删除的时候，也就不用关注监听啥状态了，传NULL
                        }
                        else
                        {   // 读取成功
                            // 写到终端
                            write(STDOUT_FILENO,buf,read_count);
                            // 发回客户端
                            write(events[i].data.fd,buf,read_count);
                        }
                    }
                }
            }

        }

    }
    return 0;
}
```